Ultracomputer Note #23
CONVERTING SCIENTIFIC CODES TO RUN UNDER THE WASHCLOTH SIMULATOR

David Korn
24 March 1981

Chapter I

INTRODUCTION

In this note I describe a set of subroutines that have been written to aid in preparing scientific FORTRAN code for WASHCLOTH simulation [1]. In addition to making code conversion simpler, these routines count machine instructions spent waiting for processor synchronization. Finally, I present the SOFTSOAP package that enables FORTRAN code prepared for WASHCLOTH simulation to run with 1 processor in a simple efficient manner.

The subroutines described in this note can be used with WASHCLOTH 81 [2] (but not with earlier versions of WASHCLOTH).

CHAPTER 2

PARALLELIZING DO LOOPS

Three FORTRAN callable functions are available to aid in the parallelization of serial DO loops. Two of the routines are described in this section and the third is described in the section on synchronization. These subroutines make it simple to convert those one and two dimensional DO loops in which loop iterations may be executed in any order.

Several of the routines in this note make use of one or more public variables, e.g. KPUB, which must be initialized to zero before parallel processing begins.

The amount of processing for each iteration need not be the same and the processors need not execute at the same rate. Whenever a processor completes a pass through the loop, it requests the next available index.

The first function to be described is used to initialize the loop indices of a loop nest providing it is not more than two levels deep. It returns the initial value for the outermost loop index. The function is called as

```
I = IDOLOOP(KPUB,J1,J2,J3,I1,I2,I3)
```

where KPUB is a public variable which must be zero prior to the earliest invocation of IDOLOOP by any processing element for this loop; J1,J2,J3 are the DO parameters for the inner loop of the nest; and I1,I2,I3 are the DO parameters for the outer loop, if there is one, but are zero in the absence of nesting. The function returns I1.

To obtain a set of indices for use within the loop nest the function NINDEX should be called at the beginning of each iteration as

```
J = NINDEX(I)
```

where I is the variable that has been initialized by the return value of IDOLOOP. If J is zero there are no more elements remaining in the loop. Otherwise I and J represent the index values for the current iteration. The value of KPUB is reset to zero by the last processing element that completes the loop. For example the one dimensional loop

```
      DO 10 J = 1,1000
              < code using index J >
   10 CONTINUE
```

would be converted to

```
      I = IDOLOOP(KPUB,1,1000,1,0,0,0)
 5    J = NINDEX(I)
      IF(J.EQ.0) GO TO 10
              < code using index J >
      GO TO 5
10    CONTINUE
```

and the value of KPUB is reset to zero when the last processor has finished the last iteration.

The two dimensional DO loop

```
      DO 20 I = 1,1000,2
              DO 10 J = 1,100,4
                      <code using I,J >
10            CONTINUE
20    CONTINUE
```

would be replaced by

```
      I = IDOLOOP(KPUB,1,100,4,1,1000,2)
10    J = NINDEX(I)
      IF(J.EQ.0) GO TO 20
              < code using I,J>
      GO TO 10
20    CONTINUE
```

The value of KPUB is set to zero by the last processing element to finish the loop. Note that IDOLOOP and NINDEX must be invoked by all of the processors in order that KPUB be reset to zero.

For non-rectangular loops, we follow the usual policy of imbedding the region in a rectangular domain and skipping the loop body for values that are not in the region of interest. For example, the two dimension triangular region described by the serial code

```
      DO 20 I = 1,1000
              DO 10 J = I,1000
                      <code using I,J >
10            CONTINUE
20    CONTINUE
```

could be replaced by the parallel code

```
      I = IDOLOOP(KPUB,1,1000,1,1,1000,1)
10    J = NINDEX(I)
      IF(J.EQ.0) GO TO 20
      IF(J.LT.I) GO TO 10
              < code using I,J >
      GO TO 10
20    CONTINUE
```

where the unwanted indices, J < I, are passed over.

To use these or any of the other subroutines or functions described in this note, add the control card:

```
LIBRARY,WASHLIB/UN=KORND.
```

when running with WASHCLOTH and

```
LIBRARY,WASHLIB=SOFTLIB/UN=KORND.
```

when running under SOFTSOAP, which is described in chapter 6. The control card sequence for a typical run is listed in section 7.

CHAPTER 3

SYNCHRONIZATION

It is often necessary to synchronize the processors at a specific point in the code. A FORTRAN callable SYNC subroutine has been written which does not return until all processing elements have invoked SYNC with the same public variable as an argument. The last processor to execute SYNC automatically resets the public variable to zero. SYNC is invoked via:

```
CALL SYNC(KPUB)
```

where KPUB is a integer public variable that must have been initialized to zero prior to the first call to SYNC.

It is important that no two successive synchronization points use the same public variable. If the same variable were used it would be possible for KPUB to become non-zero during the second synchronization before all the processing elements have passed the first synchronization and deadlock would result. To detect this error, SYNC has a builtin diagnostic feature: if it encounters the same synchronization variable on successive calls

from the same processsing element, the program aborts and prints the error message, "IMPROPER SYNCHRONIZATION".

The library also contains two routines that can be used to wait until a public variable reaches or exceeds a specified value. The first subroutine

```
WAITZ(KPUB)
```

waits until the public variable becomes zero. The second subroutine

```
WAIT(KPUB,VALUE)
```

waits until the public integer variable KPUB is greater than or equal to the integer VALUE.

A subroutine that combines the DO loop initialization step with a WAITZ at the end of the loop is provided by the function IDOSYNC. It is invoked as:

```
I = IDOSYNC(KPUB,J1,J2,J3,I1,I2,I3)
```

where the arguments to this function are the same as IDOLOOP. NINDEX is used to obtain loop indices as described earlier. When this function is used to initialize DO loops an implicit WAITZ on the public variable KPUB is made at the end of the loop.

In order to determine the synchronization overhead, a count of the instructions spent waiting is accumulated in each of the processing elements by the routines described here. A subroutine to obtain a summary of the results is described in chapter 5.

Frequently, an error in parallelizing code causes one or more processing elements to loop indefinitely waiting for synchronization. As a debugging aid, each of the synchronization routines in this library will generate a trap whenever a fixed waiting period is exceeded.

The default trap routine contained in WASHLIB responds by printing a line labelled **TIME** which contains, in order, the processor number, the clock cycle, the location of the public variable being tested in the loop, and the value of this variable.

The default waiting period is 10,000 cycles but can be reset by a call to SETWAIT with the desired number of cycles as an argument. Note, that this feature may detect long waiting periods in addition to surfacing errors.

CHAPTER 4

FURTHER COMMENTS ON SYNCHRONIZATION

The routines presented in the preceding chapters have proven valuable as an aid in scientific code conversion. However, some precaution must be observed in order to obtain high efficiency.

As an example in which a judicious use of the WAITZ subroutine can improve performance over the naive synchronization schemes indicated in chapter 3, consider the following serial code:

```
      DO 100 L = 1,100
             DO 20 I = 1,200
                    A(I) = FN1(A(I))
 20          CONTINUE
             DO 40 I = 1,300
                    B(I) = FN2(B(I))
 40          CONTINUE
100   CONTINUE
```

It is obvious that the outer loop must remain serial. The loop body contains two independent inner loops and could be replaced with an IDOLOOP for the first and an IDOSYNC for the second to ensure synchronization at the end of each iteration of the outer loop. This is illustrated by the following naive code:

```
      DO 100 L = 1,100
               J = IDOLOOP(KPUB1,1,200,1,0,0,0)
10             I = NINDEX(J)
               IF(I.EQ.0) GO TO 20
               A(I) = FN1(A(I))
               GO TO 10
20             J = IDOSYNC(KPUB2,1,300,1,0,0,0)
30             I = NINDEX(J)
               IF(I.EQ.0) GO TO 40
               B(I) = FN2(B(I))
               GO TO 30
40             CONTINUE
100   CONTINUE
```

However, we can do better by writing

```
      DO 100 L = 1,100
               CALL WAITZ(KPUB1)
               J = IDOLOOP(KPUB1,1,200,1,0,0,0)
10             I = NINDEX(J)
               IF(I.EQ.0) GO TO 20
               A(I) = FN1(A(I))
               GO TO 10
20             CALL WAITZ(KPUB2)
               J = IDOLOOP(KPUB2,1,300,1,0,0,0)
30             I = NINDEX(J)
               IF(I.EQ.0) GO TO 40
               B(I) = FN2(B(I))
               GO TO 30
40             CONTINUE
100   CONTINUE
```

In this case a wait will occur only if the A array has not been completely updated by the previous iteration when the first inner loop begins or if the B array has not completely updated by the previous iteration when the second loop begins.

To illustrate the advantage of this last code sequence, assume that FN1 and FN2 require equal time to compute and that we have 200 PEs, each executing at the same rate. Then

approximately 100 PEs would wait at each iteration of the outer loop if the naive synchronization were used, whereas, the final solution presented would result in no waiting time.

As a final example consider parallelizing the two dimensional serial loop

```
      DO 20 J = 1,N
             DO 10 I = 1,N
                    < body of loop using I,J >
10           CONTINUE
20    CONTINUE
```

without using the routines described in this note. The following solution uses a public variable KPUB and a private variable IBASE both initialized to zero, and a private variable J initialized to 1,

```
10    K = IREPAD(KPUB,1)
20    I = K - IBASE
      IF (I.LE.N) GO TO 30
             IBASE = IBASE + N
             J = J +1
             IF(J.GT.N) GO TO 40
             GO TO 20
30    < body of loop using  I,J >
      GO TO 10
40    CONTINUE
```

The value of IBASE is always (J-1)*N since it is incremented by N every time J is incremeted by 1.

A second solution, using the routines described above, is given by

```
10    K = IREPAD(KPUB,1)
      J = K/N
      I = K - J*N + 1
      IF (J.GE.N) GO TO 40
```

```
            J = J + 1
            < body of loop using I,J >
            GO TO 10
   40       CONTINUE
```

The first code requires no multiplications or divisions and is clearly more efficient for serial processing. The problem with this code for parallel processing is that the indented code is executed N times in each processor, independent of the number of processing elements. Thus for large N and O(N**2) processing elements the processing time will be dominated by the time to execute these statements and the second code is to be preferred.

CHAPTER 5

ADDITIONAL FACILITIES

As mentioned above the synchronization routines count the instruction cycles that the processors spend waiting. A subroutine has been provided to print this information. The subroutine called SUMMARY is invoked by:

```
CALL SUMMARY(KPUB)
```

where KPUB is an array of at least two public variables which must be initialized to zero. SUMMARY also prints the amount of space left in AVAIL from which the amount of AVAIL space used can be determined.

A default subroutine called GONOGO required by the 'irregular' version of WASHCLOTH is provided. When the irregular feature is used with this default GONOGO routine, processing element 1 is artificially slowed down by a factor of 8.

Finally, a default subroutine TRACE is provided so that when the WASHCLOTH tracing feature is selected, statistics on memory referencing are accumulated. These results are printed by the SUMMARY subroutine described above. When the tracing option of WASHCLOTH is selected and the default TRACE routine is used, the SUMMARY printout is augmented to include the number of instruction fetches, the number of private and public loads, the number of private and public stores, and the number of floating and integer replace-adds.

CHAPTER 6

THE SOFTSOAP SIMULATOR

The SOFTSOAP simulator is a set of FORTRAN callable subroutines that enable a user to run code intended for WASHCLOTH without getting his hands really dirty.

The SOFTSOAP package has the same entry points as WASHCLOTH. Once code has been modified to run under WASHCLOTH it can be debugged for a one processor system by running under SOFTSOAP. Since SOFTSOAP executes the code rather than emulating the instructions, it does not incur the factor of 50 times penalty associated with WASHCLOTH. In addition, since no distinction is made between public and private memory, the user need not switch to "no private" mode for diagnostic prints.

SOFTSOAP has its limitations. SOFTSOAP does not fully support the trap mechanism of WASHCLOTH due to the limited use of this feature for single processor execution. Only replace-add instructions may be trapped and in this case the instruction cycle count is only approximate.

Programs which run correctly under WASHCLOTH should run correctly under SOFTSOAP but not conversely. Therefore, SOFTSOAP is a useful tool for cleaning up errors made in code conversion, but it is no substitute for WASHCLOTH.

CHAPTER 7

CONTROL CARD SEQUENCES

In this section I present typical control card sequences for running with SOFTSOAP and WASHCLOTH.

In this example we assume that a main driver program has been written and that the source code is in a local file named DRIVER. We also assume that the source code that is to be simulated is in a local source file named TEST.

The control sequence:

```
GET,WASH=WC1S/UN=KORND.
GET,WASHLIB=SOFTLIB/UN=KORND.
LIBRARY,NYULIB.
FTN,I=DRIVER.
FTN,I=TEST,B=TESTB.
LOAD,WASH,TESTB.
LDSET,LIB=WASHLIB.
LGO.
```

can be used to run the program under SOFTSOAP. To run under WASHCLOTH change the first two cards in this sequence to

```
GETPF,OLD=WASHARC/UN=GOTTLBA.
RETRIEV,RF=WC???.
RENAME,WASH=WC???.
GET,WASHLIB/UN=KORND.
```

where ??? is the number of processing elements desired followed by the suffix B, T, I, or TI depending on whether the run is regular, traced, irregular, or traced irregular. Note that WASHLIB contains the default trap routine described in the WASHCLOTH manual.

REFERENCES

[1] Allan Gottlieb, "WASHCLOTH - The Logical Successor to Soapsuds", Ultracomputer Note #12, Courant Institute, N.Y.U., 1980.

[2] Allan Gottlieb, "WASHCLOTH 81", Ultracomputer Note #21, Courant Institute, N.Y.U., 1981.